V. 12. Рекурсия: перебор вариантов.

Перебор: циклы большой вложенности.

Задача о расстановке ферзей

Начнём с классической переборной задачи - задачи о расстановке ферзей. Задача такая.

Имеется шахматное поле - квадратная таблица размером NxN. Конечно, совсем уж в классической задаче N=8, но давайте уж сразу чуть-чуть обобщим. Требуется расставить N ферзей так, чтобы никакие два из них не били друг друга. Если совсем уж отказаться от шахматной терминологии, то задача состоит в том, чтобы отметить в таблице N клеток так, чтобы никаие две из них не находились ни на одной горизонтали, ни на одной вертикали и ни на одной прямой, напрвавленной под углом 45^o к границам таблицы.

Сразу замечаем, что, поскольку никакие два ферзя не лежат на одной горизонтали, а их количество равно количеству горизонтальных рядов, то в каждом ряду должен находится ровно один ферзь. А дальше идея ясная: перебираем все возможные расстановки ферзей по одному в каждом ряду и проверяем, удовлетворяет ли она условиям задачи - не бьёт ли какой-то ферзь другого ферзя. Каждый ферзь может находится в любом столбце, значит ставим каждого ферзя поочерёдно во все столбцы - вот вам и перебор. Ну, понятно, что реализуется это N-кратным циклом. Что-то вроде

```
    for (ставим ферзя в первом ряду во все колонки от 1-й до N-й) {
        for (ставим ферзя во втором ряду во все колонки от 1-й до N-й) {
            for (ставим ферзя во третьем ряду во все колонки от 1-й до N-й) {
                ...         ...         ...
                for (ставим ферзя в N-ом ряду во все колонки от 1-й до N-й) {
                    if (никакие два ферзя не бьют друг друга) {
                        Обрабатываем полученную позицию - печатаем, подсчитываем, и
т.д.
                    }
                ...         ...         ...
                }
            }
        }
```

```
    }
```

Немного поразмыслив (если сразу мы к этому не пришли), построим вот такую естественную модификацию алгоритма:

```
    for (ставим ферзя в первом ряду во все колонки от 1-й до N-й) {
        for (ставим ферзя во втором ряду во все колонки от 1-й до N-й) {
            if второй ферзь бьёт первого {переходим к следующей колонке}
            for (ставим ферзя во третьем ряду во все колонки от 1-й до N-й) {
            if 3-й ферзь бьёт кого-то из первых двух {переходим к следующей колонке}
                ...         ...         ...
                for (ставим ферзя в N-ом ряду во все колонки от 1-й до N-й) {
                    if N-й ферзь бьёт кого-то из первых N-1 ферзей {
                        переходим к следующей колонке
                    }
                    обрабатываем полученную позицию - печатаем, подсчитываем, и т.д.
                    }
                }
            }
        }
    }
```

Код, соответствующий первому варианту алгоритма, я не стану приводить: во-первых эти коды мало чем отличаются, во-вторых, предложенная во втором варианте оптимизация совершенно очевидна, так что можно считать второй вариант начальным :)

пример queens_cyclic.go

```
package main

import "fmt"

const N = 5

type queen struct {
              col int
              row int
           }

func abs(x int) int {
    if x >= 0  {
```

```
        return x
    } else {
        return -x
    }
}

func Connected (q1, q2 queen) bool {
    return q1.col == q2.col ||
           q1.row == q2.row ||
           abs(q1.col - q2.col) == abs (q1.row - q2.row)
}

func Conflict (qs []queen, q queen) bool {
    for _, q2 := range qs {
        if Connected (q, q2) { return true }
    }
    return false
}

func main() {
    var Queens [N]queen
    for col0:= 0; col0 < N; col0++ {
        Queens[0] = queen{col0, 0}
        if Conflict(Queens[:0], Queens[0]) { continue }
        for col1:= 0; col1 < N; col1++ {
            Queens[1] = queen{col1, 1}
            if Conflict(Queens[:1], Queens[1]) { continue }
            for col2:= 0; col2 < N; col2++ {
                Queens[2] = queen{col2, 2}
                if Conflict(Queens[:2], Queens[2]) { continue }
                for col3:= 0; col3 < N; col3++ {
                    Queens[3] = queen{col3, 3}
                    if Conflict(Queens[:3], Queens[3]) { continue }
                    for col4:= 0; col4 < N; col4++ {
                        Queens[4] = queen{col4, 4}
                        if Conflict(Queens[:4], Queens[4]) { continue }
                        for _, q := range Queens {
                            fmt.Printf("%c%d ", q.col+'a', q.row + 1)
                        }
                        fmt.Println()
                    }
```

```
                }
            }
        }
    }
}
```

У меня хватило здоровья на 5 ферзей - на пятикратный цикл, или, иначе говоря, на цикл вложенности 5.
А что делать с классической шахматной доской 8x8? Писать цикл вложенности 8? Ужас. Просто ужас. А если N=10? Получается, что для каждого N надо писать отдельную программу. Как-то нехорошо получается. Как-то это неправильно выглядит…

Что же делать? Задуматься, понятное дело. Что такое цикл вложенности N? Как его выполнить? Очень просто – несколько раз (сколько требует самый внешний цикл) выполнить цикл вложенности (N-1). Рекурсия просто прёт из этой фразы. Ну, так и не будем сопротивляться. Терминальный случай – цикл без вложений (т.е. цикл вложенности 1), и, как это обычно бывает с терминальным случаем, с ним всё понятно.

`пример queens_recursive.go`

```
package main

import "fmt"

const N = 5

type queen struct {
               col int
               row int
            }

func abs(x int) int {
    if x >= 0  {
        return x
    } else {
        return -x
    }
}

func Connected (q1, q2 queen) bool {
    return q1.col == q2.col ||
```

```
            q1.row == q2.row ||
            abs(q1.col - q2.col) == abs (q1.row - q2.row)
}

func Conflict (qs []queen, q queen) bool {
    for _, q2 := range qs {
        if Connected (q, q2) { return true }
    }
    return false
}

var Queens [N]queen

func Search(n int) {
    if n == 0  {
        for _, q := range Queens {
            fmt.Printf("%c%d ", q.col+'a', q.row + 1)
        }
        fmt.Println()
        return
    }
    for col:= 0; col < N; col++ {
        Queens[N-n] = queen{col, N-n}
        if Conflict(Queens[:N-n], Queens[N-n]) { continue }
        Search(n-1)
    }
}

func main() {
    Search(N)
}
```

Вспомогательные функции, понятное дело, не изменились, также, как и вывод позиции. В функции Search(n) аргумент означает степень (степень, а не глубину) вложенности цикла и, одновременно, сколько ещё ферзей осталось поставить. Единственная вольность по сравнению с написанным алгоритмом допущена в том месте, где идёт обработка терминального случая. Функции Search(n) мы даём в терминальном случае вызвать себя с n=0, а уже этот случай и считаем терминальным. Довольно логично, учитывая, что n - это сколько ферзей осталось поставить.

Задача о разбиении числа на сумму четырёх квадратов

Задача уже появлялась в прошлом занятии в домашнем задании. Если дети её делали - очень хооршо, ляжет на разогретое место, это всегда приятно. Задача такая, процитирую прошлый план:

```
Вход: натуральное число n, n<=2000000000. Разбить его на сумму четырёх квадратов цел
ых неотрицательных чисел. Например,
    20 = 3^2^ + 3^2^ + 1^2^ + 1^2^
    или
    20 = 4^2^ + 2^2^ + 0^2^ + 0^2^,

    100 = 7^2^ + 7^2^ + 1^2^ + 1^2^
    или
    100 = 9^2^ + 3^2^ + 3^2^ + 1^2^
    или
    100 = 8^2^ + 6^2^ + 0^2^ + 0^2^
    или
    100 = 8^2^ + 4^2^ + 4^2^ + 2^2^
    или
    100 = 7^2^ + 5^2^ + 5^2^ + 1^2^
    и т.д.
Найти все решения.
```

В задании имелось в виду рекурсивное решение, но давайте по-простому, напишем тупо решение с четверным циклом, благо 4 - это немного.
программа 4squares_cycle1.go

```go
package main

import "fmt"

const size = 4

func search(sum int) {
    for x1:= 0; x1*x1 <= sum; x1++ {
        for x2:= 0; x2*x2 <= sum; x2++ {
            for x3:= 0; x3*x3 <= sum; x3++ {
```

```
                for x4:= 0; x4*x4 <= sum; x4++ {
                    if x1*x1 + x2*x2 + x3*x3 + x4*x4 == sum {
                        fmt.Println(x1, x2, x3, x4)
                    }
                }
            }
        }
    }
}

func main() {
    n:= 10
    search(n)
}
```

А теперь будет просто грешно не воспользоваться опытом, полученным при работе с 8 ферзями, и не переделать эту программу в рекурсивную

программа 4squares_rec1.go

```
package main

import "fmt"

const size = 4

func search(sum int, amount int, result []int) {
    if amount == 0  {
        if sum == 0 {
            fmt.Println(result)
        }
        return
    }
    for x:= 0; x*x <= sum; x++ {
        search(sum - x*x, amount - 1, append(result, x))
    }
}

func main() {
    n:= 10
    search( n, size, make([]int, 0) )
}
```

Запуская эти программы, получаем

```
0 0 1 3
0 0 3 1
0 1 0 3
0 1 3 0
0 3 0 1
0 3 1 0
1 0 0 3
1 0 3 0
1 1 2 2
1 2 1 2
1 2 2 1
1 3 0 0
2 1 1 2
2 1 2 1
2 2 1 1
3 0 0 1
3 0 1 0
3 1 0 0
```

Ой, ужас-ужас, повторы. Что делать? Хранить все результаты и отсекать повторы? Ну, можно и так, но как-то это означает добавить лишнюю работу (программе, а не себе, т.е. себе, конечно, тоже, но мы об эффективности и качестве кода). Причём это двойная добавка - мы делаем и дополнительную обработку, и обрабатываем лишние данные, те самые, которые мы находим повторно. Гораздо эффективнее избежать обработки лишних данных - тех, которые уже фактически были. Как это сделать? Например, давайте договоримся, что мы будем обрабатывать (и подавать на обработку) только неубывающие четвёрки чисел: `x1<=x2<=x3<=x4` . И тогда повторы отсекаются автоматом. В общем, код показывает, что имеется в виду:

функция search из программы 4squares_cycle2.go

```
func search(sum int) {
    for x1:= 0; x1*x1 <= sum; x1++ {
        for x2:= x1; x2*x2 <= sum; x2++ {
            for x3:= x2; x3*x3 <= sum; x3++ {
                for x4:= x3; x4*x4 <= sum; x4++ {
                    if x1*x1 + x2*x2 + x3*x3 + x4*x4 == sum {
                        fmt.Println(x1, x2, x3, x4)
```

```
                    }
                }
            }
        }
    }
}
```

Остальная часть программы не изменяется, так что ни к чему её здесь приводить.

И, соответственно, в рекурсивном варианте функция search выглядит так:
функция search из программы 4squares_rec2.go

```
func search(sum int, amount int, result []int) {
    if amount == 0  {
        if sum == 0 {
            fmt.Println(result)
        }
        return
    }
    var start int
    if len(result) == 0 {
        start = 0
    } else {
        start = result[len(result)-1]
    }
    for x:= start; x*x <= sum; x++ {
        search(sum - x*x, amount - 1, append(result, x))
    }
}
```

Продолжим оптимизацию кода. На рекурсивной функции это особенно хорошо видно: если мы раскладываем число N на K слагаемых, то наименьшее из них не превосходит N/K. Получаем в итоге код:
функция search из программы 4squares_rec3.go

```
func search(sum int, amount int, result []int) {
    if amount == 0  {
        if sum == 0 {
            fmt.Println(result)
        }
        return
```

```
    }
    var start int
    if len(result) == 0 {
        start = 0
    } else {
        start = result[len(result)-1]
    }
    for x:= start; x*x*amount <= sum; x++ {
        search(sum - x*x, amount - 1, append(result, x))
    }
}
```

Нерекурсивный вариант:

функция search из программы 4squares_cycle3.go

```
func search(sum int) {
    for x1:= 0; x1*x1*4 <= sum; x1++ {
        for x2:= x1; x2*x2*3 <= sum - x1*x1; x2++ {
            for x3:= x2; x3*x3*2 <= sum - x1*x1 - x2*x2; x3++ {
                for x4:= x3; x4*x4 <= sum - x1*x1 - x2*x2 - x3*x3; x4++ {
                    if x1*x1 + x2*x2 + x3*x3 + x4*x4 == sum {
                        fmt.Println(x1, x2, x3, x4)
                    }
                }
            }
        }
    }
}
```

Но не будем на этом успокаиваться. Ведь последний (самый внутренний) цикл совершенно не нужен - мы там проверяем, можно ли разложить число на сумму одного ! квадрата, достаточно просто проверить, является ли это число точным квадратом. Это легко сделать. Смотрим код:

программа 4squares_cycle4.go

```
package main

import (
    "fmt"
    "math"
```

```
)

const size = 4

func PerfectSquare (n int) (sqrt int, is bool) {
    sqrt = int( math.Round( math.Sqrt( float64(n) ) ) )
    return sqrt, sqrt*sqrt==n
}

func search(sum int) {
    for x1:= 0; x1*x1*4 <= sum; x1++ {
        for x2:= x1; x2*x2*3 <= sum - x1*x1; x2++ {
            for x3:= x2; x3*x3*2 <= sum - x1*x1 - x2*x2; x3++ {
                if x4, ok:= PerfectSquare(sum - x1*x1 - x2*x2 - x3*x3); ok {
                        fmt.Println(x1, x2, x3, x4)
                }
            }
        }
    }
}

func main() {
    n:= 500
    search(n)
}
```

Функция PerfectSquare как раз и проверяет, является ли число точным квадратом, и, если да, то возвращает его квадратный корень.

И рекурсивная версия:
программа 4squares_rec4.go

```
package main

import (
    "fmt"
    "math"
)

const size = 4
```

```
func PerfectSquare (n int) (sqrt int, is bool) {
    sqrt = int( math.Round( math.Sqrt( float64(n) ) ) )
    return sqrt, sqrt*sqrt==n
}

func search(sum int, amount int, result []int) {
    if amount == 1  {
        if x, ok:= PerfectSquare(sum); ok {
            fmt.Println(append(result, x))
        }
        return
    }
    var start int
    if len(result) == 0 {
        start = 0
    } else {
        start = result[len(result)-1]
    }
    for x:= start; x*x*amount <= sum; x++ {
        search(sum - x*x, amount - 1, append(result, x))
    }
}

func main() {
    search(500, size, make([]int,0))
}
```

В общем, сюжет интересный и несложный. Тем и хорош.

Перебор: циклы переменной вложенности.

В двух предыдущих задачах вложенность цикла мы знали заранее. Хотя в задаче с ферзями она могла быть произвольной в определённых рамках, но всё равно она была известна уже на этапе разработки программы (design-time). А что делать, если вложенность цикла становится известной только во время выпонения программы (run-time)? Тут, кажется, выход только один - рекурсия. Давайте посмотрим на простейшую такую ситуацию.

Генератор комбинаций (сочетаний) - цикл с

неопределённой заранее вложенностью

Имеется множество из N различных элементов. *Сочетанием (комбинацией) из N по K элементов* называется набор из K элементов, выбранных из этого множества. Наборы, отличающиеся только порядком следования элементов (но не составом), считаются одинаковыми.

Выпишем все сочетания из N=6 элементов по K=2: [1 2], [1 3], [1 4], [1 5], [1 6], [2 3], [2 4], [2 5], [2 6], [3 4], [3 5], [3 6], [4 5], [4 6] и [5 6]. В принципе, идея уже ясна, но можно выписать и комбинации из N=5 по K=3: [1 2 3], [1 2 4], [1 2 5], [1 3 4], [1 3 5], [1 4 5], [2 3 4], [2 3 5], [2 4 5], [3 4 5]. Причём пишем именно в таком порядке. Дети скорее всего почувствуют, что речь идёт о лексикографическом порядке, даже если они это явно не смогут формулировать. Очень хочется не свалиться на хаотичное выписывание комбинаций. В крайнем случае, загоняем поток занятия в лексикографический порядок явно. Особенно прозрачна идея цикла вложенности K на примере с K=2 – что-то вроде составить список всех игр в однокруговом турнире, каждая команда должна встретиться один раз с каждой другой. В общем, тут довольно легко прийти к чему-то такому:

```
// Алгоритм: реализуем цикл вложенности K
// for i1:= 1; i <= N-(K-1) {
//     for i2:= i1+1; i <=  N-(K-2) {
//         for i3:= i2+1; i <=  N-(K-3) {
//           . . .      . . .      . . .
//             for iK:= i(K-1)+1; To N-(K-K) {
//                 Print (i1, i2, ..., iK)
//             }
//           . . .      . . .      . . .
//         }
//     }
// }
```

Конечное значение каждой переменной цикла определяется тем, что вслед за ней должно остаться место для следующих за ней переменных - переменных циклов большей глубины.

Ситуация очень похожа на ту, которая сложилась с избавлением от повторов в задаче про разбиение на сумму четырёх квадратов, только там мы делали каждое следующее число не меньше предыдущего - повторы были разрешены, а здесь повторы запрещены, так что каждое следующее число должно быть строго больше предыдущего.

Всё понятно, но проблема в том, что вложенность цикла мы заранее, в момсент написания

кода, не знаем. Что делать? А что такое цикл вложенности К? Так мы же это уже знаем и делали. Повторю, чтобы его выполнить надо просто несколько раз (сколько требует самый внешний цикл) выполнить цикл вложенности (К-1). И опять, с терминальным случаем – цикл без вложений (т.е. цикл вложенности 1, он же глубины К) - всё понятно. А нетерминальный случай должен сколько–то там раз вызывать цикл на единицу меньшей вложенности (ну, или на единицу большей глубины).

Ага. Значит нам для каждого из К циклов надо знать

1. глубину вложенности цикла и
2. диапазон изменения переменной цикла: начальное значение и конечное значение.

В нашем случае

```
в цикле глубины 1 переменная цикла изменяется от     1 до N-K+1
в цикле глубины 2 переменная цикла изменяется от  i1+1 до N-K+2
в цикле глубины 3 переменная цикла изменяется от  i2+1 до N-K+3
в цикле глубины 4 переменная цикла изменяется от  i3+1 до N-K+4
```

и т.д.

Легко видно, что конечное значение переменной цикла = `N - K + глубина_цикла` , так что достаточно передавать только две (из трёх) величин: начальное значение переменной цикла и глубину цикла (можно, конечно, вместо глубины передавать конечное значение переменной цикла). И всё! Вот вам программа, которая делает в точности то, что мы сказали. Практически тупая запись всего вышесказанного.

`Пример combinations.go`

```go
package main

import "fmt"

var N, K int

var P []int //глобальный "массив"

func Cikl(CurrentPos int, FirstItem int) {
    // CurrentPos - степень вложенности цикла,
    // FirstItem - стартовое число очередного цикла
    for i := FirstItem; i <= N-(K-CurrentPos); i++ {
        P[CurrentPos-1] = i
```

```
        // Цикл вложенности K - последний.
        if CurrentPos == K {
            // Печатаем комбинацию (сочетание)
            fmt.Println(P)
        } else {
            Cikl(CurrentPos+1, i+1)
        }
    }
}

func main() {
    for {
        fmt.Print("Enter N: ")
        fmt.Scanln(&N)
        fmt.Print("Enter K: ")
        fmt.Scanln(&K)
        if K>0 && N>=K { break }
    }
    // Алгоритм: реализуем цикл вложенности K
    // for i1:= 1; i <= N-(K-1) {
    //     for i2:= i1+1; i <=  N-(K-2) {
    //         for i3:= i2+1; i <=  N-(K-3) {
    //           . . .      . . .      . . .
    //             for iK:= i(K-1)+1; To N-(K-K) {
    //                 Print (i1, i2, ..., iK)
    //             }
    //           . . .      . . .      . . .
    //         }
    //     }
    // }
    P = make([]int, K, K)
    Cikl(1, 1)
}
```

Но, в общем-то, пример, при всей его важности, весьма несложный, проблем с ним особых не предвидится, так что не будем на нём зависать.

И продолжим ситуацией, когда вложенность цикла не только становится известна только во время выполнения, но и

Вложенность цикла изменяется в процессе выполнения программы

Конечно же, речь пойдёт не о степени вложенности цикла, а о глубине рекурсии. Да, можно и эту ситуацию разрешить с помощью вложенных циклов, но это уже бужет совсем что-то несуразное. А задачу рассмотрим такую.
На шахматной доске расставить наименьшее количество ферзей так, чтобы все клетки находились под боем хотя бы одного из них Считаем, что ферзь не бьёт ту клетку, на которой он находится.

Ситуация во многом напоминает ситуацию с генератором комбинаций. В самом деле, набор ферзей - это некоторая комбинация клеток. И нас интересуют комбинации, обладающие некоторым свойством - свойством бить все клетки доски. И мы точно также добавляем ферзей к имеющемуся списку, и точно также двигаем последнего ферзя, начиная с клетки, следующей за предыдущим ферзем, и до последней клетки доски. Главное отличие - если список ферзей уже настолько длинный, что присоединять к нему ферзя нет смысла - всё равно уже имеющийся наилучший результат он не улучшит, то мы не работаем больше с этим списком и возвращаемся на предыдущий уровень рекурсии (вложенности цикла). Всё остальное изложено в коде - он довольно подробно комментирован.
программа aggressive_queens.go

```
package main

import "fmt"

const n = 8

type
    cell struct {
        row, col int
    }

func (c cell) Connected(c2 cell) bool {
    // Соединены ли клетки c и c2 ходом ферзя?
    if c == c2 { return false }
    return    (c.row == c2.row) || (c.col == c2.col) ||
            (abs(c.row-c2.row) == abs(c.col-c2.col))
}
```

```
func (c cell) Print() {
    fmt.Printf("%c%d ", c.col+'a', c.row+1)
}

func (c cell) Next() cell {
    // Возвращает клетку, следующую за клеткой c.
    // Направление движения: вдоль столбца - увеличиваем строку,
    // в конце столбца переходим на нижнюю клетку следующего столбца
    if c.row < n-1  {
        return cell{c.row + 1, c.col}
    } else {
        return cell{0, c.col + 1}
    }
}

func (c cell) Terminal() bool {
    // Верно ли, что c - последняя клетка на доске?
    return c.Next().col == n
}

func Success(list []cell) bool {
    // Верно ли, что все клетки доски находятся
    // под боем какого-то ферзя из списка list
    for row:= 0; row < n; row++ {
        for col:= 0; col < n; col++ {
            ok := false
            for _, c:= range(list) {
                if (cell{row, col}).Connected(c) {
                    ok = true
                    break
                }
            }
            if !ok {
                return false
            }
        }
    }
    return true
}

var result []cell  // здесь храним текущее наилучшее решение
```

```
func search(list []cell) {
    if list[0].col == n-1 && list[0].row == n - len(list) {
    // терминальный случай: последняя комбинация ферзей,
    // дальше двигаться некуда
        return
    }
    if len(list) >= len(result)-1 {
    // добавлять ферзей бессмысленно - улучшить результат не удастся
        return
    }
    // last - последий ферзь ф текущем списке
    last:= list[len(list) - 1]
    // добавляем ещё одного ферзя
    for c:= last.Next(); !c.Terminal(); c = c.Next() {
        if Success( append(list, c) ) {
        // если новый ферзь делает список таким, что все
        // все клетки находятся под боем, то этот список
        // улучшает текущий результат - запоминаем его
            result = append(list, c)
            return
        }
        // если новый ферзь не далает список таким,
        // что все клетки находятся под боем,
        // то пытаемся добавить ещё ферзей
        search( append(list, c)    )
    }
}

func main() {
    // Начальное решение - заполняем ферзями весь нижний ряд
    for i:= 0; i< n; i++ {
        result = append(result, cell{0, i} )
    }
    // Поиск начинается со списка из одного ферзя,
    // стоящего в первой клетке - клетке {0, 0}
    search([]cell{cell{0,0}})
    // Печать результата
    for _, c := range (result) {
        c.Print()
    }
```

```
    fmt.Println()
}

func abs(x int) int {
    if x<0 {
        return -x
    } else {
        return x
    }
}
```

Остаётся только заметить, что размер доски можно сделать и не константой, а переменной, и вводить её в начале программы.

Ну, и напоследок, совсем простая рекурсия, но эта задача совершенно классическая и даёт начало целому потоку задач, вытекающих из неё. Некоторые брызги из этого потока долетят и до нас в виде задач для самостоятельных штудий.

Перебор-перебор. Задача о рюкзаке.

Условие задачи следующее. Имеется рюкзак, который может поместить груз в N килограммов (или других единиц веса). Имеется набор грузов. В решении они названы bricks - кирпичи, но это неважно. Считаем, что веса кирпичей не повторяются - все кирпичи весят по-разному. Требуется набить рюкзак вплотную: выбрать набор кирпичей, суммарный вес которых ровно N килограммов. Мы напишем программу, которая находит все решения.

Условие о неповторяемости весов кирпичей нужно для того, чтобы избежать повторения решений. Задача с повторяющимися весами кирпичей - в задачах для самостоятельной работы - задача “Монеты”.

Решение совсем простое. Берём первый кирпич. Мы его можем либо положить в рюкзак, если в нём хватает места, либо не класть в рюкзак. В любом из двух случаев после этого остётся решить ту же задачу для всех кирпичей, исключая первый. Терминальный случай - кирпичей не осталось, понятное дело.

Код простой, прозрачный, думаю особо комментировать его не придётся.
программа backpack.go

```
package main
```

```
import "fmt"

func solve (bricks []int, rest int, solution []int) {
    if rest == 0 {
    // нашли решение
        fmt.Println(solution)
        return
    }
    if len(bricks) == 0 {
    // решения не нашли, а кирпичи кончились...
        return
    }
    // Решаем:
    // либо мы берём первый кирпич, ...
    if rest >= bricks[0] {
        solve(bricks[1:], rest - bricks[0], append(solution, bricks[0]))
    }
    // ... либо не берём
    solve (bricks[1:], rest, solution)
}

func main() {
    bricks:= []int{7, 11, 24, 11, 3, 28, 4, 6, 12} // кирпичи
    carrying:= 48 // грузоподъёмность сумки
    solve(bricks, carrying, make([]int, 0))
}
```

Задачи для самостоятельной работы

Сначала, как и обещали, серия задач вокруг задачи о рюкзаке.

Рюкзак-1. “Весы”. Имеется набор гирь, известны их веса. Имеются чашечные весы, гири можно ставить на обе чаши весов. Можно ли, пользуясь этими гирями, взвесить предмет весом ровно N кг? Взвесить - это означает доказать, что он весит именно N кг: поставить предмет на какую-то чашу весов и с помощью имеющихся гирь уравновесить весы. Если можно, то привести какой-нибудь вариант это сделать.

Рюкзак-2. “Монеты”. В кошельке имеются монеты нескольких номиналов. Известно, сколько монет каждого номинала есть в кошельке. Требуется определить, можно ли заплатить имеющимися монетами заданную сумму, и, если да, то как именно это сделать.

Рюкзак-3. “Ограбление”. Грабим магазин. У нас есть сумка заданной грузоподъёмности. В магазине есть несколько предметов, про каждый из них известны его вес и его стоимость. Задача состоит в том, чтьобы унести из магазина товаров на как можно бОльшую сумму. Но чтобы при этом все они влезли в сумку. Набивать сумку полностью, как это было в задаче о рюкзаке, не обязательно.

И ещё несколько задач на перебор

Перебор-1. Вот такая головоломка. Имеется комплект из 2N карточек (N не очень большое, не больше 15-16, ну, может, до 18, всё зависит от скорости работы программы). На двух из них написано число 1, на двух - число 2, на двух - число 3 и т.д. Выложить карточки в ряд так, чтобы между двумя единицами находилась ровно одна карточка, между двумя двойками - ровно две карточки, меджду двумя тройками - ровно три карточки и т.д. Например, один из вариантов решения для N=4: 41312432.

Перебор-2. Генерировать случайную расстановку кораблей для игры в “морской бой”. Кратко о правилах: комплект кораблей - прямоугольников - один корабль 1x4, два корабля 1x3, три корабля 1x2 и четыре корабля 1x1 на до разместить на игровом поле; игровое поле - квадрат 10x10. Корабли не должны иметь общих точек (в том числе граничных или угловых).

Перебор-3. Имеется N команд (N - четное число). Задан массив

```
var A [N][N] bool
```

В A[i][j] отмечено, играли ли уже между собой i-я и j-я команды. Разумеется, A[i][j] = A[j][i], A[i][i] = false. Составить какое-нибудь (случайное!) расписание на следующий тур (если это возможно) - в туре должны играть все команды, причем ранее встречавшиеся пары не должны играть между собой.

Перебор-4. Имеется таблица NxN. Некоторые клетки содержат числа из диапазона от 1 до N, некоторые клетки - пустые (ну, например, обозначены 0). N – не очень большое, порядка 4-8. Надо так заполнить пустые клетки, если это возможно, так, чтобы каждая строрка и каждый столбец содержал полный комплект чисел от 1 до N.

Перебор-5. Генерировать (случайно!, т.е. каждый раз другой) маршрут обхода шахматным

конём шахматной доски. Можно для простоты размер доски уменьшить, только не очень сильно, до 5-6, а то обхода не будет вовсе существовать. Точнее – надо пройти конём по всем клеткам, зайдя в каждую ровно один раз. Маршрут не обязан быть замкнутым (возвращаться в начальную клетку не обязательно). Типичнейший перебор с возвратом. Возврат здесь не только логический (в дереве перебора), а даже физический (конь возвращается)

Ну, и напоследок очень показательная задача на оптимизацию рекурсии

“Последовательность”. Вычислить значение функции f(n), определенную для всех натуральных чисел, следующим образом:
f(1) = 1,
f(n) = f(n/2)+f(n/3)+ … +f(n/n) при n>1.
Написать тупую рекурсивную реализацию прямо из определения функции очень легко. Но надо это сделать для дальнейшего сравнения. А уже потом заняться оптимизацией этой реализации. Задача модельная в том смысле, что оптимизация здесь видна отчётливо - заменять сложение одинаковых слагаемых на умножение, например, f(11) = f(5) + f(3) + f(2) + f(2) + f(1) + f(1) + f(1) + f(1) + f(1) + f(1) = f(5) + f(3) + 2*f(2)* + *6*f(1).
Причём хорошо бы написать и сравнить ещё и реализации без мемоизации и с мемоизацией.

Хватит пока. Потом, глядишь, и добавим что-нибудь, задач на перебор - море.